SAVY: STREAM ALGORITHM VISUALIZATION FOR YOU

Presented By:

Nicolas Scarrci

Completion Date:

May 11, 2011

Approved By:

Assistant Professor Andrew McGregor, Computer Science

Associate Professor Ramesh Sitaraman, Computer Science

ABSTRACT

Title: **SAVY: Stream Algorithm Visualization for You**
Author: **Nicolas Scarrci, Computer Science**
CE Type: **Independent Capstone Project**
Approved By: **Andrew McGregor, Computer Science**
Approved By: **Ramesh Sitaraman, Computer Science**

This software, SAVY, is intended to provide its users an environment in which to become acquainted to stream algorithms, experiment with stream algorithms, and eventually to implement, analyze, and improve their own stream algorithms. For introducing users to stream algorithms SAVY provides an implementation of the following algorithms: the "Count-Min sketch" as presented by Graham Cormode and S. Muthukrishnan, a sparse spanner algorithms as presented by Michael Elkin, and a maximum matching approximation algorithm as presented by Andrew McGregor. SAVY allows users to enter their own data and observe how each algorithm reacts. Users can also choose to allow the software to generate random data so that users can view the execution of the algorithms over time. SAVY also exposes an API to allow users to add their own algorithms to the software. To encourage the analysis of approximation algorithms such as the Count-Min sketch SAVY provides the absolute answers for comparison.

1 INTRODUCTION

Since the execution of an algorithm is effectively the same as series of operations performed on data structures over time, they are difficult to represent statically (e.g. with lecture slides or handouts.) One often overlooked and consistently underused technique is algorithm visualization. Algorithm visualization is the act of visually displaying the data structures that an algorithm manipulate,s and then updating them as the algorithm executes. As a field, algorithm visualization has not seen much attention from the computer science community. In their 2006 paper "Algorithm Visualization: A Report on the State of the Field," Shaffer et al. [10] state that there simply are not many algorithm visualizations available on the internet, and that many of those which are available are of low quality.

In addition to addressing the state of the field, they also address problems commonly found in algorithm visualizations. For example, they state that many visualizations have little to no user interaction. In some cases this means that the user cannot enter the data that the algorithm uses. In the most extreme cases this means that the visualization is actually an animation with no dynamic components. This directly contradicts research which shows that the effectiveness of a visualization as a tool to assist understanding is directly related to how much they user interacts with it. As Grissom et al. [4] showed, students who viewed visualizations outperformed students who did not view visualizations. A more interesting result of the study was that students who "responded" to the visualizations, which is to say answered questions about what the visualization was doing while watching it, outperformed all other groups.

Shaffer et al. also showed that the majority of the visualizations that exist are on low level topics. Of the data they collected in 2006 over 38% of the visualizations elucidated sorting algorithms, and over 17% elucidated search structures. Furthermore, some of the visualizations did not actually visualize the execution of the algorithm and instead would simply show the result of a series of operations. One example of this is a visualization of a heap structure which shows the updated heap after an insertion or a deletion, but does not visualize how the heap is restructured. In their concluding

remarks they state: "we ... simply need more implementors[*sic*] to produce more quality visualizations. And we need to encourage them to provide visualizations on under-represented topics."

2 LITERATURE REVIEW

While the problem we are attempting to solve is the lack of quality visualizations with useful features, our goal is to provide a single environment in which visualizations can be created, explored, analyzed, mutated, and eventually shared. Similar things have been attempted before at various times in the past with varying goals and various levels of success.

Algorithm visualization has a closely related sister field called software visualization. Software visualization is concerned with the workings or performance of individual programs. Software visualization suites are generally more concerned with tracking memory allocation and efficiency of a running program than they are with individual algorithms. A very good example of this is IBM's Jinsight project which focuses on "Object-oriented visualization for performance tuning and program understanding," "Pattern visualization to study repetitive behavior and explore data structures," and "Memory leak diagnosis" [5].

A good survey of algorithm visualization systems can be found in [6]. To summarize the relevant parts of that work, the major systems in algorithm visualization are BALSA and TANGO. BALSA is a program visualizer that was developed at Brown University. BALSA was originally designed for use with PASCAL and would follow the execution of a program by highlighting which line of code was currently being evaluated. For ease of use, BALSA would reformat the source code to make it more readable. Since the original BALSA project many smaller derivative systems have been created, all of which appear proprietary to Brown University. TANGO was also developed at Brown, although somewhat later. The main difference between BALSA and TANGO is that TANGO visualizes program execution by having the program establish events which are reported to the system as the program executes. These events are then mapped to

animations. In this way the system avoids constantly updating choosing instead to only update when something changes. This method is clearly more conducive to programs where sparse visual updates are expected.

In 2000 Naps et al. released a paper discussing their project JHAVÉ, an "environment designed to actively engage students in web-based algorithm visualizations" [8]. JHAVÉ itself is not actually a visualization engine. Instead it is a wrapper which provides extra functionality to stand-alone visualization code. JHAVÉ interfaces with executing algorithms through the use of a script file. As an algorithm executes it writes visualization commands to this script file, which is then used by JHAVÉ to actually produce a manipulable visualization. This design requires that the algorithm finish executing before JHAVÉ can be used. By taking this approach, JHAVÉ requires that users wait for algorithms with high order runtimes, or very large datasets to finish executing before the user can see the visualization.

Despite this one feature which inhibits user interaction with the executing algorithm, JHAVÉ does a very good job of promoting user interaction. This includes the support of "start and stop questions," as suggested by Grissom et al., as well as "context-sensitive documentation" which explains what the algorithm is doing at each point in the visualization (ostensibly controlled through script files.) One of JHAVÉ's unique features is the presence of a "rewind" feature which allows users to "undo" previous actions undertaken by the visualization.

JHAVÉ is also unique in that it implements a client-server architecture. In JHAVÉ users connect to a server which exposes a list of available algorithms. Users then request a certain algorithm and (presumably) send the dataset on which they would like it to be run to the server. The server executes the algorithm and then sends the script file generated to the client which the client then parses and presents on the screen. This is a particularly interesting decision. While it does take advantage of the server's (most likely) superior computing power it also introduces the problems inherent to the client-server architecture (e.g. intermittent availability and introducing a single point of failure) without necessarily gaining anything.

The literature overwhelmingly suggests that one unified engine which supports the addition of individual algorithms at a later date is needed as it would allow the standardization of algorithm visualization. While JHAVÉ makes a strong showing toward this end it accidentally prohibits algorithms from certain areas of research. Moreover, the presence of a single engine which is linked to a large repository of algorithm visualizations would provide the community with a single point of focus. It is imperative that this engine allow for custom data in order to engage the users. It is also important that it be easy for individuals to produce new algorithm visualizations so that users can contribute to the base of available algorithms.

3 THE SAVY SOFTWARE

SAVY is an algorithm visualization suite which is currently being developed by Nicolas Scarrci at the University of Massachusetts Amherst. SAVY consists of base java classes which provide the framework for algorithm visualizations. To aid users in generating new visualizations SAVY provides abstract super classes as well as a user manual available at [9] which explains how to properly extend these base classes. SAVY currently ships with three example visualizations.

3.1 Contributions to the Field

As stated by Shaffer et al. "we need to encourage ... visualizations on under-represented topics." One such under-represented topic is stream algorithms. Stream algorithms are characterized by having limited (often sequential) data access and very small (polylogarithmic) space bounds. Because of this, stream algorithms are particularly well suited to large datasets. With commercial giants like Google and Apple gathering datasets of unprecedented size one would expect stream algorithms to quickly gain standing in the standard computer science curriculum. With that in mind, SAVY ships with three new visualizations which are based on relatively new stream algorithms (published between 2005 and 2011.) In addition SAVY also includes one entirely new stream algorithm variant.

SAVY is also unique in that it concerns itself with algorithm development analysis and testing instead of processor use or memory leak diagnosis. Unlike JHAVÉ, SAVY visualizes algorithms while it executes them, allowing the user to run their visualizations on large datasets. This is particularly useful when developing stream algorithms. By visualizing the algorithm as it executes, instead of running the visualization after the algorithm completes, SAVY also lends itself to online algorithms which can be queried for their solutions at any point during their execution, not just at the end. By allowing on the fly data entry SAVY increases interaction with the user who can now experiment at will instead of having to halt the visualization, send an amended dataset to a server, and then waiting for a result.

SAVY is also amenable to testing any-time algorithms. Because the architecture is such that visualizations allow their algorithms to process when they receive a new update it is possible to delay these updates to simulate receiving them in real time. By simulating the types of update traffic that certain algorithms would experience in their intended deployment it is possible to make empirical comparisons independent of theoretical results. Though development is ongoing, a base implementation of delayed data is currently available in SAVY.

3.2 Architecture

SAVY does not utilize the internet and therefore does not have a network architecture. Herein we discuss the architecture of the software itself. For reference a diagram has been included in appendix C. While SAVY is provided as both an applet and an application (for accessibility purposes,) we will refer to them both in this section as SAVY, since the bulk of their code is the same.

When SAVY is launched the user is presented with the algorithm launcher. The purpose of this window is to allow the user to launch multiple different visualizations without needing to open multiple windows or launch multiple copies of the application. A screenshot of this window is available in appendix D.1. The algorithm launcher allows the user to select from a list of included visualizations, see a description of the

visualization, set the parameters of the visualization and then launch it in a separate window. When a new visualization is started its launcher is initialized.

The launcher serves as an intermediary between the implemented algorithm, the user, and the data that the algorithm is processing. By separating the algorithm implementation and the launcher implementation SAVY encourages users to create more general launchers which can be used to visualize multiple algorithms (e.g. a graph launcher which provides functionality unique to graphs.) This separation also prevents users from unintentionally changing the functionality of the algorithm when they are intending to change the visualization of the algorithm only. Screenshots of the three included visualization launchers can be found in appendices D.2, D.3, and D.4.

3.3 Implemented Algorithms

The first algorithm that is included with SAVY is the Count-Min sketch of Graham Cormode and S. Muthukrishnan in their paper "An improved data stream summary: the Count-Min sketch and its applications." To understand how the algorithm works it is important to first understand the problem that it is trying to solve. Imagine that we are being handed flash cards which have some number of a certain type of shape. In order to receive the next flash card it is necessary that we discard the flash card that we are holding. Once we have finished looking at all of the flash cards we will be asked how many of each shape we have been shown.

Since computers don't have hands or eyes we formulate the problem in the following way. We represent each flash card as a pair of numbers (x,n) where x represents which type of shape is on the card and n represents how many copies of that shape were on the card. We then present these flash cards to the computer as a stream (which is to say that the computer can't look at the old flashcards.) Now let us imagine that the stream of flash cards is so long that it is infeasible to have the computer remember all of them. Instead of maintaining a count for each shape, we can instead maintain a 'sketch' of the data.

Our solution is to maintain a matrix of counters. The height and width of this matrix

are dependent on how accurate we want our approximation to be and how often we are willing to have the algorithm make a mistake. When we receive an update (x,n) we update some cell in each row of the matrix by n (where the cells we update are dependent on x.) To answer a question about how many of shape x we have seen we look at the cells that we update for shape x. Because these cells may have also been updated by other shapes we cannot simply return the value of one of the cells. Since we know that we only make positive updates (there cannot be -5 squares on a page,) we can assume that the cell with the smallest value is the cell which was updated by the fewest other shapes, so we return the value in this cell as our estimate.

PSEUDOCODE

for all $update(x, n)$ **do**

  for all row **do**

    $Table[H[row](x)][row] \leftarrow Table[H[row](x)][row] + n$

  end for

end for

For the implementation of this algorithm see CountMin.java available at [9]. For proofs regarding the algorithm as well as its original description see [1]

Before we move on from this algorithm it is important to explain how we chose which cells to update for each shape. For each row of the table we have a hash function which is randomly produced. Our hash functions belong to the family H(x) = (ax+b mod p) mod w, where a $\in$ [1,...,p-1], b $\in$ [0,...,p-1], p is a prime larger than the range of x, and w is the width of the matrix. This family of hash functions helps to minimize cell collisions between different values of x.

The next algorithm that we include is the 2t-1 spanner presented by Michael Elkin in his paper "Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners." The problem solved by a sparse spanner is a common problem in computer science, especially with ever increasing datasets. Imagine that we have a graph, but this graph is too large to hold at once, so we want to instead remember

a subset of the edges of the graph such that we do not inflate any path lengths by more than a factor of 2t-1. Since the graph is too large to hold at once, we must process the edges as a stream of data.

Elkin suggests that in order to generate a spanner we can maintain a small subgraph centered on each node. As our edges come in we will grow these subgraphs, and eventually connect them. When an edge (u,v) comes in we check the subgraph coming out of u and check to see if it already contains a path to v. If it does then we can ignore the edge (u,v), but if it doesn't then we add (u,v) to our subgraph if it is not already large. If our subgraph on u is already large then we stop adding edges and instead check to see if the subgraph on u is connected to the subgraph on v. If it is then we can ignore the edge, if it is not then we add this edge to a separate subgraph. Once we have read all of the data we return the union of these subgraphs.

DATA STRUCTURES

A stretch factor t.

A list of levels, L(x) which represent the level of x, or distance from the root.

A list of base values, B(x) which represent the identity of the tree that x belongs to.

A list of trees, T(x) which represent the subgraphs which we grow out of nodes x.

A list of sets, X(x) which represent edges which connect the tree that x is in to other trees.

PSEUDOCODE

for all edges (u,v) **do**

$wlog$ assume u is closer to its root than v

if u belongs to a small enough tree **then**

L(v) $\leftarrow$ L(u)+1

B(v) $\leftarrow$ B(u)

T(v) $\leftarrow$ T(v) + (u,v)

end if

if T(u) is not connected to T(v) **then**

X(v) $\leftarrow$ (u,v)

end if

end for

For the implementation of this algorithm see SparseSpanners.java available at [9]. For a proof of the stretch guarantee or the original description of the algorithm see [2].

The final algorithm that is included in SAVY is Andrew McGregor's Maximum Matching algorithm from his paper "Finding graph matchings in data streams." This algorithm solves the problem of finding the set of edges with maximum weight such that no two edges share a vertex. While the maximum matching problem has an exact answer computable in polynomial time in the standard model (unlimited data access,) only an approximation is known in the streaming model. The idea behind this algorithm is rather simple, accept a new edge if it is better than the edges that it conflicts with.

This approach is not quite sufficient. Imagine that our graph was simply a line edges with slightly increasing weights (perhaps 1+ϵ, 1+2ϵ, etc.). Clearly the best matching that we can make is to pick every other edge in the line. If our current algorithm were to process them in order it would accept the first, then reject it to accept the second, then reject that to accept the third, and so on. In the end this algorithm would have only the last edge, which is far from optimal. To combat this we require that for an incoming edge to replace old edges it must be at larger than the sum of the conflicting edges by at least a multiplicative factor of (1 + γ), where γ is selected by the user. This improves the approximation ratio greatly.

DATA STRUCTURES

A multiplicative increase factor γ.

A list of selected edges.

PSEUDOCODE

for all updates (u,v) **do**

for all selected edges that conflict with (u,v) **do**

if weight(u,v) $>$ weight(conflicted edges)(1+γ) **then**

selected edges $\leftarrow$ selected edges - conflicted edges

selected edges ← selected edges + (u,v)

end if

end for

end for

For the implementation of this algorithm see WeightedMatching.java available at [9]. For proofs regarding the algorithm or its original description see [7].

This algorithm is implemented with optional randomized rounding. In the case that randomized rounding is selected the edges are rounded into weight classes $(1+\gamma)^{\delta}$, $(1+\gamma)^{\delta+1}$, and so on, where δ is selected at random from (0,1]. This allows the algorithm to check whether an edge belongs to a sufficiently large weight class when replacing old edges instead of checking if it is larger by a significant factor. This greatly simplifies the analysis of the algorithm, but requires that the effects of the randomized rounding be evaluated. The application of randomized rounding is borrowed from "Improved approximation guarantees for weighted matching in the semi-streaming model" by Leah Epstein et al. For independent result regarding the approximation ratio with rounding see appendix A.

3.4 UI and Visualization Considerations

From a distance, algorithm visualization may appear to be rather cut and dry, but this is not so, there are all sorts of things which must be considered. For example, in visualizing the Count-Min sketch it may seem obvious that to visualize the algorithm the matrix should be displayed as a matrix and the cells chould be updated in real-time. While this covers the basics it does not make the visualization clear. Without highlighting the cells that are changing and slowing the whole visualizatiion down it is difficult for the user to understand what the algorithm is doing. Even with the cells highlighted the use of the hash functions is difficult to understand unless they are presented visually as well. Similarly, query answering is entirely unclear unless the cells that are being compared are highlighted.

The other two algorithms invited a whole lot of other visualization issues peculiar to

graphs. The first issue was figuring out how to generate a graph that would be visually pleasing to a human. This included things like not allowing duplicate edges. Whenever a duplicate edge was added to the graph it appeared instead as though the original edge was being updated. I also noticed that once an algorithm ran for a while it would become cluttered, obscuring the algorithm's execution. This is because the graphs contained a large number of edges, and stream algorithms usually maintain only a small subset of those edges.

It also quickly became apparent that color could be used to bring a 'third dimension' to any graph. Unlike how color was used in the Count-Min sketch to highlight changes that are interesting for the user to notice, or suggest how a process works, graph algorithms can use color to encode additional data about the graph. In the sparse spanner problem different colors are used to mark the different subgraphs, as well as the "cross edges" which connect them. For the maximum matching problem the darkness of a color was used to present data about how 'old' an edge was. In this specific case the edges that were bumped from the matching were darkened over time to imply that they were less relevant to current execution the algorithm.

Though it does not occur in any of the three visualizations in SAVY, additional data could be encoded either in text or in shape. As more visualizations are implemented more unique and useful ways of encoding information visually will be discovered, and hopefully applied to old visualizations which were not sufficiently clear. As a field the issue of algorithm visualization is far from solved.

4 FUTURE WORK

After 'finishing' SAVY I find that there are still a lot of features that I would like to add to the software. For this project I focused more on the teaching aspect of the software than I would like. I feel that novel research is still to be done on the algorithm design and analysis possibilities of visualization software. In the future I would like to provide more support for design and analysis. For example, the current software requires that a user add to the source code and then recompile it in order to add their

own algorithm/visualization. This is cumbersome, and while the software exposes an API (see appendix B,) it is still unreasonable to assume that most users will be willing to edit the source code. By requiring source code edits I am also preventing users from easily sharing or modifying their algorithms. To change this I would like to add a tool which allows the user to write pseudocode describing how their algorithm works and then, by adding special marks to the pseudocode describe how the would like the execution of the algorithm to be visualized.

I would also like to add the ability to load pre-made sets of data into each visualization. This would let users run repeatable experiments as well as allow users to create a set of datasets for use as a benchmark. In addition to this I would like to allow for the user to select two algorithms and run them in a side by side comparison mode that with highlight whenever the two algorithms make different decisions. When comparing two algorithms it would also be useful to provide information about the amount of space each algorithm is using as well as how fast each algorithm is running. To assist users in generating data, especially in the case of graph algorithms it would be useful to present a tool that would allow them to draw a graph directly on the screen, instead of forcing them to enter edges in the form (u,v). I feel that it would also be useful to expose a new superclass which users could extend when writing algorithms that run in the standard model. This would allow users to qualitatively compare the performance of their approximation algorithms to the actual best solutions.

I feel that this software has a lot of potential for use in the field if these suggested features are added. By increasing the ability of users to add algorithms it is my hope that I can create a small online community where user-generated visualizations can be submitted to a repository and then downloaded by other users. Ideally I could then release bundles of visualizations for use by teachers. Eventually I could provide a message board service to allow users to provide feedback on specific visualizations as well as post request an algorithm be added.

REFERENCES

[1] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the Count-Min sketch and its applications. *J. Algorithms* 55, 1 (April 2005), 58-75. http://dx.doi.org/10.1016/j.jalgor.2003.12.001

[2] Michael Elkin. 2011. Streaming and fully dynamic centralized algorithms for constructing and maintaining sparse spanners. *ACM Trans. Algorithms* 7, 2, Article 20 (March 2011), 17 pages. http://doi.acm.org/10.1145/1921659.1921666

[3] Leah Epstein, Asaf Levin, Julián Mestre and Danny Segav. 2009. Improved appocimation guarantees for weighted matching in the semi- streaming model. In *Symposium on Theoretical Aspects of Computer Science.* 347-358.

[4] Scott Grissom, Myles F. McNally, and Tom Naps. 2003. Algorithm visualization in CS education: comparing levels of student engagement. In *Proceedings of the 2003 ACM symposium on Software visualization* (SoftVis '03). ACM, New York, NY, USA, 87-94. http://doi.acm.org/10.1145/774833.774846

[5] The Jinsight Group at IBM. IBM Research - Programming Languages and Software Engineering - Research Areas. http://www.research.ibm.com/jinsight

[6] Benjamin Kurtz. Softviz: A Runtime Software Visualization Environment. *Master's Thesis at WPI*

[7] Andrew McGregor. 2005. Finding Graph Matchings in Data Streams. *Approximation, Randomization and Combinatorial Optimization. Algorithms and Techniques* (Lecture Notes in Computer Science.) 3624, 611-612. http://dx.doi.org/10.1007/11538462_15

[8] Thomas L. Naps, James R. Eagan, and Laura L. Norton. 2000. JHAVÉ—an environment to actively engage students in Web-based algorithm visualizations. *SIGCSE Bull.* 32, 1 (March 2000), 109-113. http://doi.acm.org/10.1145/331795.331829

[9] Nicolas Scarrci. 2011. SAVY Home Page. http://www.cs.umass.edu/~nscarrci/SAVY

[10] Clifford A. Shaffer, Matthew Cooper, and Stephen H. Edwards. 2007. Algorithm visualization: a report on the state of the field. *SIGCSE Bull.* 39, 1 (March 2007), 150-154. http://doi.acm.org/10.1145/1227504.1227366

APPENDIX

A APPROXIMATION RATIO OF MAXIMUM MATCHING WITH ROUNDING

For this algorithm we have rounded the input by a method called randomized geometric grouping. All edge weights are separated into classes from $(1+\gamma)^{p+\delta}$ to $(1+\gamma)^{p+1+\delta}$ where p $\in \mathbb{Z}$ and $\delta \in (0,1]$. Edge weights are always rounded down.

Theorem A.1. *With rounding the maximum matching algorithm presented achieves an approximation ratio of* $\frac{\gamma-1}{2(1+\gamma)ln(1+\gamma)}$ *which is maximized at ≈.187 for* $\gamma \approx 4.36$.

When one edge bumps one or two other edges from the matching we know that the weight of this edge is greater than the combined weight of the edges that were discarded by a factor of:

$$\frac{(1+\gamma)}{2} \geq 1 \tag{1}$$

Let w(M) be the weight of our matching, and let T(M) be the edges which were removed from M. Next we prove two lemmas about w(M).

Lemma A.0.1. $w(T(M)) \leq \frac{2}{\gamma-1} w(M)$

Proof. If we separate the trail of an edge into sets C_i such that set C_i was bumped by set C_{i+1}, then we can say that each set is at least a factor $\frac{1+\gamma}{2}$ larger than the set that it bumps. If we let k be the index of the last set that was bumped then we can write $\frac{(1+\gamma)}{2} w(C_k) \leq w(M)$. If we sum over the sets C_i then we can say: $\frac{(1+\gamma)}{2} \Sigma_{i \leq k} w(C_i) \leq \Sigma_{i \leq k} w(C_i) + w(M)$. Noting that $\Sigma_{i \leq k} w(C_i) = w(T(M))$ we can write $\frac{1+\gamma}{2} w(T(M)) \leq w(T(M)) + w(M)$. By simple algebra we have $\frac{1+\gamma}{2} w(T(M)) \leq w(T(M) + w(M) \Rightarrow \frac{\gamma-1}{2} w(T(M)) \leq w(M) \Rightarrow w(T(M)) \leq \frac{2}{\gamma-1} w(M)$. □

Lemma A.0.2. $w(OPT) \leq w(T(M)) + 2w(M)$

Proof. If we think of OPT as a set of edges, $o_1, o_2, ...o_k$ then we can say that each edge o_i in OPT was either added to M, or prevented by at least one corresponding edge e_i. If we consider charging the costs of the edges in OPT to the edges in M we

can write a relationship between OPT and M. It may be the case however, that some edges in OPT were"rejected" because of edges that were in M at the time, but were later rejected themselves. This means that we will also have to charge costs to T(S). If an edge e_i is responsible for the rejection of edge o_i in OPT then we can charge it for w(o_i). Since we don't know w(o_i) we can approximate it by w(e_i) which we know is at least as much as w(o_i). We can extend this reasoning to the case where two edges are responsible for the rejection of o_i (since $w(e_i) + w(e_j) > w(o_i)$). We can approximate this total weight by w(T(M)) (since this weight must be at least as much as the actual sum). Once the algorithm terminates the only edges which can be charged for more than one edge of OPT are those that are in the matching M. Since we don't know which edges in M caused rejections of edges in OPT we can approximate the sum of the weights of these edges by the weight of the matching, w(M). This allows us to write $w(OPT) \leq w(T(M)) + 2w(M)$. □

By combining our two lemmas we can write:

$$w(OPT) \leq \frac{2}{\gamma - 1} w(M) + 2w(M) = \frac{(2\gamma)}{(\gamma - 1)} w(M) \tag{2}$$

Because we have rounded our input based on a randomized variable we cannot reason about the actual values of w(M), so we reason about the expected value instead.

$$E_\delta[OPT'(\delta)] \leq \frac{2\gamma}{(\gamma - 1)} E_\delta[w(M)] \tag{3}$$

To deal with the rounding we need one more lemma.

$$\frac{(1+\gamma)\ln(1+\gamma)}{\gamma} E_\delta[OPT'(\delta)] \geq OPT \tag{4}$$

Let us consider the expected value of the weight of a single edge. If we can prove $\frac{(1+\gamma)\ln(1+\gamma)}{\gamma} E_\delta[w_\delta(e)]$ then the lemma follows. Let us consider an edge with unrounded weight w(e) = $(1+\gamma)^{p+a}$, where $0 \leq a < 1$. Now we integrate $(1+\gamma)^{p+\delta}$ for $0 \leq \delta \leq$ a and a $< \delta < 1$. For $\delta \leq$ a we round down to $(1+\gamma)^{p+\delta}$ and for $\delta >$ a we round down

to $(1+\gamma)^{p-1+\delta}$.

$$E_\delta[w_\delta(e)] = \int_0^a (1+\gamma)^{p+\delta} d\delta + \int_a^1 (1+\gamma)^{p-1+\delta} d\delta = w(e)(1-\frac{1}{(1+\gamma)})\frac{1}{ln(1+\gamma)} \quad (5)$$

We can combine equations 3, and 4 to get a statement about OPT in terms of w(M)

$$w(OPT) \leq \frac{(1+\gamma)\ln(1+\gamma)}{\gamma}\frac{2\gamma}{(\gamma-1)}E_\delta[w(M)] = \frac{2(1+\gamma)\ln(1+\gamma)}{\gamma-1}E_\delta[w(M)] \quad (6)$$

The analysis regarding the effects of rounding the input is heavily borrowed from [3].

□

B USER MANUAL

B.1 Adding Your Own Algorithms

Now that you've become acustomed to the visualizations that ship with SAVY perhaps you'd like to start adding your own visualizations? The followinig section details the API that SAVY exposes to you.

B.1.1 Creating Your Own SlowData

Disclaimer

The first step to understand MySlowDataQueue is to understand that it should not be altered in any way. MySlowDataQueue should provide enough of an abstraction that any changes that need to be made can be made by creating your own SlowData that extends MySlowData. If you find it absolutely necessary to edit the way that MySlowDataQueue works please extend the class so as to avoid breaking all of the built in algorithms, as well as any user defined algorithms that you may have acquired.

MySlowDataQueue contains three lists of MySlowData, posted, others, and permanentData. One important feature of MySlowDataQueue is that it allows for the simulation of real-time acquisition of data. It does this by moving MySlowData from the others list, which is not accessible from outside of the class to the posted list which provides access to the first element of the list (via MySlowDataQueue.poll()). This movement is done by the update() method which is called periodically by each algorithm. Whenever an item is moved to the posted list it is also moved to the permanentData list. Data items that are added to the permanentData list are never removed. This allows you as the analyst to view and process the data that the algorithm has seen so far. This is useful when comparing an estimate to the actual value (as is done in the Count-Min sketch).

MySlowDataQueue also provides the utility function randomize(). This function does NOT generate random data. That is done on a per algoritm basis, since MyS-

lowDataQueue does not know very much about the structure of the MySlowData that your algorithm will be using. The randomize function simply randomizes the order of the data in the others list. Arguably this should only be done once directly after data has been generated or loaded, though it may be called at arbitrary times during the algorithm's execution.

B.1.2 Understanding MySlowData

The MySlowData class requires that each implementing subclass override the default toString() method with some meaningful toString() method. The MySlowData class also provides the default expired() method which is used to test if an item should be added to the posted queue or not. MySlowData also provides a utility method for setting an expiration date in the future called fixDate(long). This simply allows you to 'delay' a data item by some number of seconds. This method should be overridden in an implementing subclass if finer granularity than seconds is desired.

B.2 Implementing an Algorithm

B.2.1 Writing a Constructor

The constructor is where the initialization of all of the data structures you intend to use happens. Note that this is NOT where the generation of random data, or the loading of data from files should happen. This IS however the place where data structures that are randomized should be generated.

B.2.2 Understanding churn()

The churn method simply calls a series of overrideable functions in the logical order. Churn calls:

- getCanvas()
- canvas.data.updateData()
- canvas.onUpdateSuccess()

- canvas.onUpdateFailure()
- canvas.otherProcessing()

getCanvas()

getCanvas() is an abstract function which simply returns the paintable component that you have used for your visualization. Because you must override getCanvas() you can place your canvas anywhere you would like.

updateData()

updateData() simply tells MySlowDataQueue to post any new items and then returns the oldest new item.

onUpdateSuccess()

This is the code that gets executed every time a new data item is posted. This is where the majority of your algorithm should go. Don't forget to include a call to repaint() at the end, or your algorithm won't appear to do very much!

onUpdateFailure()

This is the code that gets executed if the environment checks for new data, but no data is found. This can be used for performing processing that the algorithm does during "down time." (ex enriching data structures, or sorting)

otherProcessing()

This function is executed regardless as to whether there is new data or not. Generally speaking this method is more for things like timing execution and debugging, but it can be overridden if required by the algorithm.

B.2.3 Courtesy Methods

These methods are not strictly required should be implemented as a courtesy to the user of your visualization.

getInfo()

This method returns a block text which is presented to the user when they consider selecting your visualization. This text should explain which problem your visualization

solves as well as anything unique about the specific algorithm that it implements. This text should also explain what the starting parameters are as well as how to control the visualization.

getLauncherDisplay()

This method returns a container which contains all of the buttons and fields necessary to retrieve input from the user. If for example you are visualizing an approximation algorithm which has a specifiable confindence you can use this method to display a field in which the user can enter this parameter. The ordering and display of your fields are used as is and are not changed by the software. This allows you to access them directly when calling the init(...) method of your launcher class.

B.2.4 Understanding the Paint Method

paint()

This is arguably the most important function that you need to override. This is where all of the visualization goes on. Specifically this is where code for your visualization goes (as other components like buttons are repainted automatically.)

B.2.5 Data Initialization

genData()

This function is an abstract function which should be overridden to generate random data. Note that this function is only intended for generating the data in MySlowDataQueue. Any other data, for example a list of random nodes, should be generated when data structures are initialized during init(...).

genBase()

This is a safety function which simply generates a blank copy of whatever component you decide to use for visualization. This is intended for use in initialization and error handling, but can be overridden by a blank function if desired.

B.3 Creating a Launcher

B.3.1 Understanding the Constructor

Launchers currently work in two phases. The first is where the launcher allows for input from the user before the visualization begins. This code exists in the constructor. I recommend setting default values and setting up error handlers which catch invalid input and substitute default values.

B.3.2 Understanding init(....)

The second phase is the initialization phase. Once the user has started a visualization the init method should be called. This method is in charge of processing the user's input, creating an instance of your algorithm class, and setting up the screen to reflect whichever commands the user should now have access to. Don't forget to remove the buttons you added in the constructor!

B.3.3 Writing an actionPerformed(e) method

The actionPerformed(e) method is inherited from ActionListener and is required to make your buttons work. One point that may seem confusing is that there is one actionPerformed method for both the buttons added in our launcher's constructor, and the buttons added in init(...). Though it is not necessary for a visualization I highly recommend copying the code for "Step" "Finish" and "End," as these actions provide the user control over the execution of the visualization.

B.4 Tips and Tricks

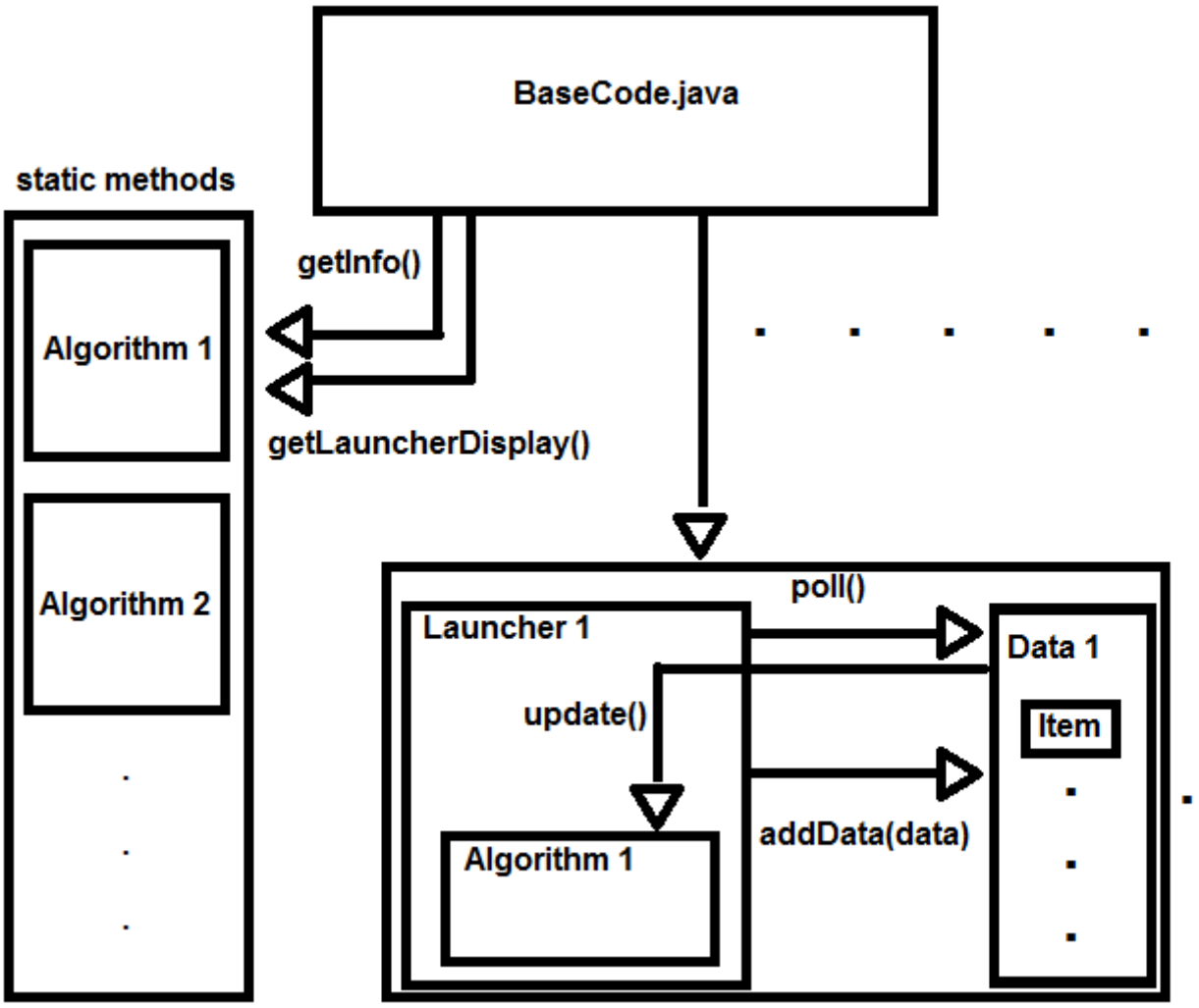
B.4.1 Graphs

The first tip I would suggest for graphs is to use geometric graphs. These are the easiest to visualize, and generally the easiest to understand.

The second tip I would suggest is to remember that color allow essentially a "third" dimension, with "darkness" representing a fourth. The third tip I would suggest is to generate edges in an efficient manner. For example, if you have already randomly

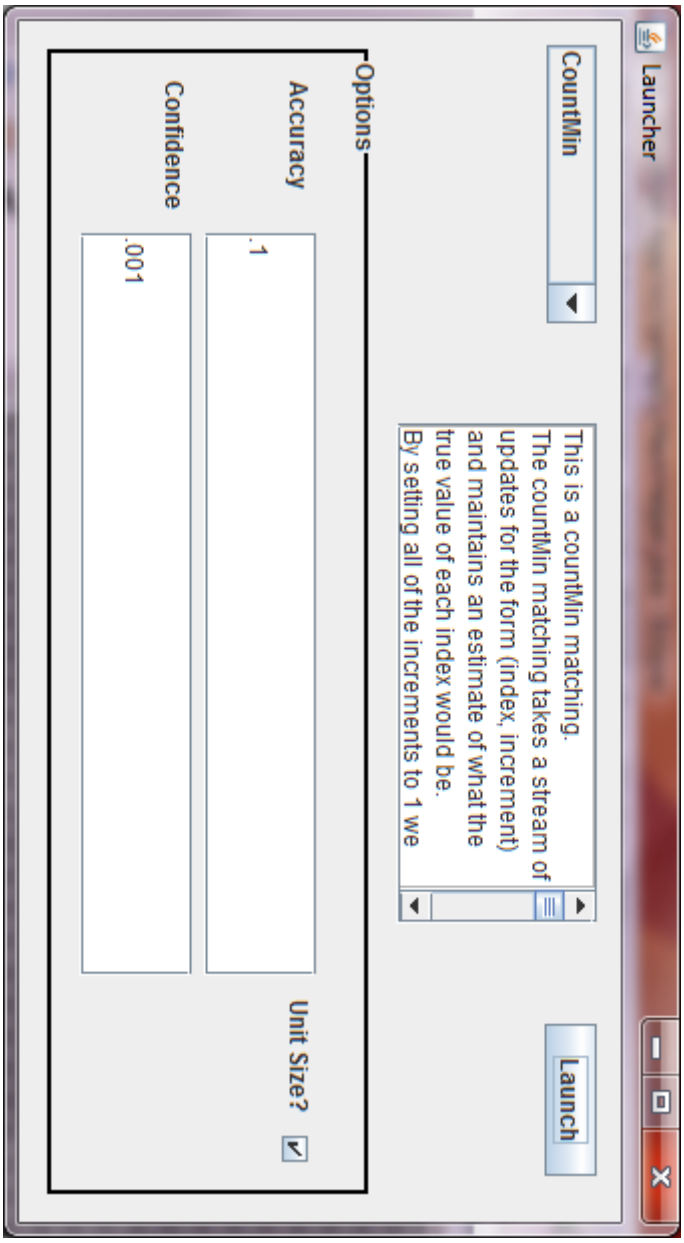
generated your nodes is it any less random to simply select all edges of certain lengths? If you choose this path you can speed up your algorithm by "boxing" your nodes. If you were to draw a grid across your plane each square would represent a box. By selecting a certain length for the sides of the boxes, you can ensure that the only edges that can exist within certain bounds must be either between nodes in the same box, or nodes in adjacent boxes. Thus in order to "generate" edges you can simply walk across these boxes performing (with high probability) a small number of comparisons for each of the at most 4 pairs of boxes. This also prevents the possibility of duplicate edges. This method is employed in both SparseSpanner.java and WeightedlMatching.java.

C ARCHITECTURE DIAGRAM



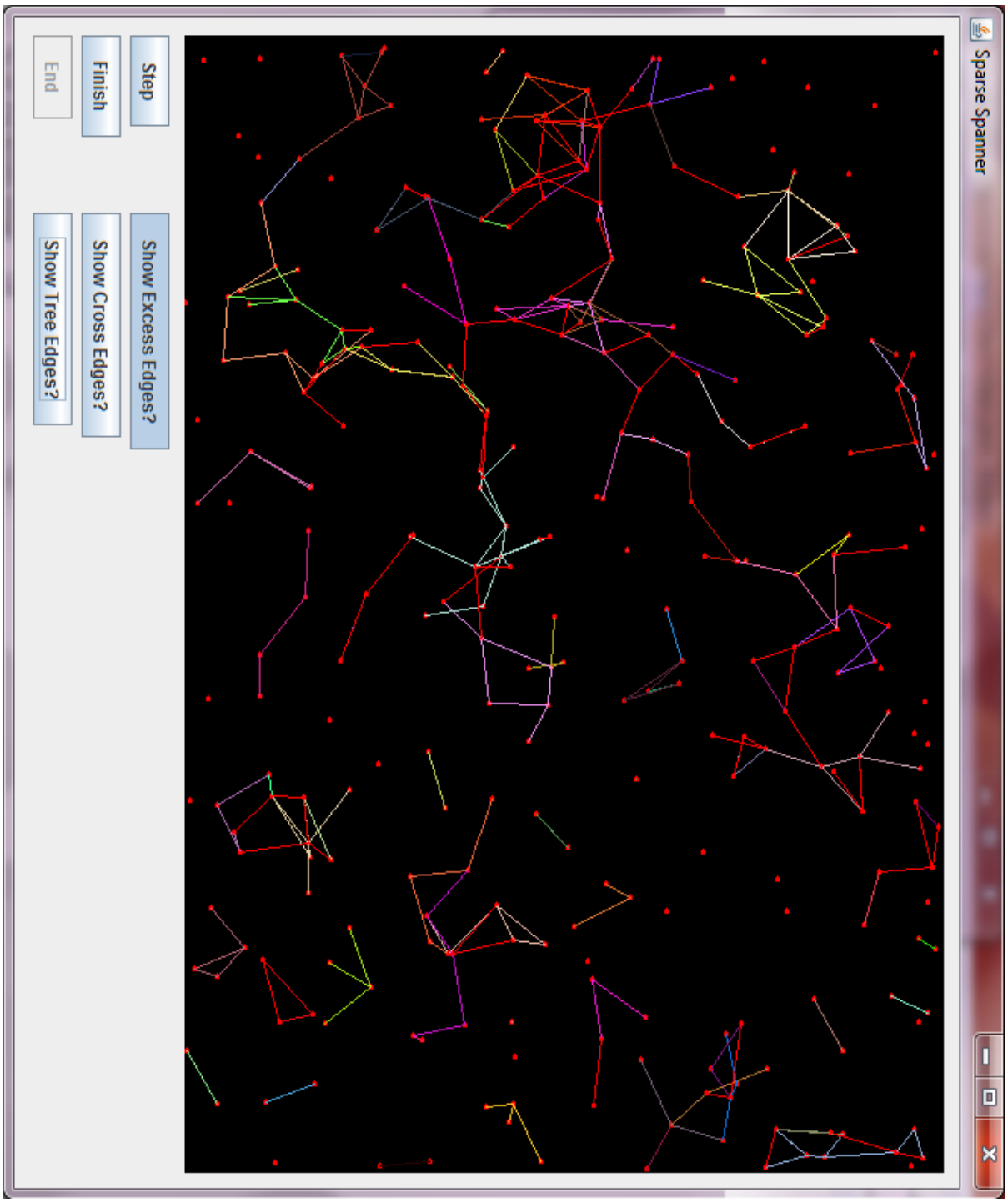
D SCREENSHOTS

D.1 Algorithm Launcher



D.2 Count Min Launcher

D.3 Sparse Spanner Launcher



D.4 Maximum Matching Launcher